Lecture 1: Introduction

An **operating system** is a piece of software that acts as an intermediary between users/applications and computer hardware.

- Provides abstraction; hides hardware architecture details from the applications
    - This deals with diversity of hardware components, transparancy
- Manages the sharing of data and hardware resources among applications and users
- Optimizes performance via resource management

It must provide the following services:

- Process management; creation, deletion, scheduling, suspension, resumption, synchronization, IPC
- Memory management; allocation, deallocation, efficient utilization under heavy workload
- I/O management; device-driver interface
- File management; manipulation, mapping onto 2nd storage, manage free space, disk scheduling
- Error detection
- Protection; concurrent processes may not interfere
- Security
- Accounting

Operating systems are **interrupt driven**. An **interrupt** is a signal from a device *to* the CPU that transfers control to an interrupt service routine. The **interrupt vector table** contains the addresses of such handlers. The OS must preserve the CPU state before handling. Similarly, a **trap** or **exception** is a *software-generated* interrupt, either fro a software error or a system call.

An OS has two modes of execution: **user mode** (for user code) and **kernel mode**. This way, the OS protects itself and other system components: we can prevent apps from modifying data in locations of other applications, as we require kernel mode to do such an operation, which first checks whether the operation is valid. A system call is required to switch between the modes.

There are different motivations for having an OS:

1. Dealing with diversity - allowing applications to run on any brand of hardware, and different system architectures
2. Transparency - the same code can be compiled for different systems, virtual memory ensures the programmer need not concern themselves with memory constraints or locations in physical memory
3. Virtualization - simple, abstract, logical model of the system, programs can consider themselves the sole user of resources
4. Support shared functionality - common to most programs, accessed through system calls
    a. Often accesses by programs via an API, combining many system calls to implement tasks
5. Portability - programs written using an OS or API can run on any system supporting that OS/API.
6. Concurrency - share the system fairly between multiple active processes and users (no starvation)

Lecture 2: Processes, threads and scheduling (double lecture)

2.1: Processes

A **process** is a *program in execution*, with its own **context of execution** and resources. They don't share memory. They're defined by their *text* (code), *stack* (local variables, return addresses), *heap* (store data with yet unknown size), *data* (global vars with known size), and information about the current state of execution. The **process control block** (**PCB**) contains a process' context of execution. Having concurrent processes requires **context switching**: save the state of the process to be suspended, and reload this state when it is to be resumed.

Parent processes can create child processes, with either full/partial/no resource sharing. Parents can execute concurrently with their children, or wait until children terminate. In Linux, the child process is a copy of the parent process (copy of the address space), which can then be overwritten. Child processes become independent after creation, but start execution at the point of creation!.

- fork() - creates identical copy of the caller process, returns a process id (0 for the child process, and child-pid for the parent)
    - allows branching of parent and child after creation
- execlp() - overwrites calling process with its argument program, code following this call is never reached (except in error)
- wait() - blocks until one child exits
- waitpid() - blocks until a specific child changes its state

When a process calls exit(), the status is returned to the parent if it was waiting. Then, the OS takes the process' resources away. Alternatively, the parent may terminate execution of children (**abort**). If no parent is waiting, the child process is a **zombie**. If the parent terminated without invoking wait(), the child process is an **orphan**.

Processes allow **parallelism**: with multiple cores, multiple processes can make simultaneous progress. Also allows **concurrency**, where the same resource is *time-shared* between processes to improve responsiveness. If a process is waiting for data or a resource, the kernel can *schedule another process* to make efficient use of system resources. Do note that **process switching** has overhead from mode & context switching (due to saving/restoring states).

2.2: Threads

A **thread** is a *dispatchable unit of work within a process*, sharing their memory address space (but have their own regs and stack). Creation of, switching & communicating between threads is much faster due to shares address space. Care must be taken to protect threads from other threads when making changes to shared data. We distinguish between **user threads** (thread libraries) and **kernel threads**. We require a mapping between user- and kernel threads (*more information in the book*).

- Many-to-one - multiple user threads to one kernel thread. Only one user thread can access the kernel at a given time
- One-to-one - allows concurrency, but the kernel must manage all threads impacting performance
- Many-to-many - allows concurrency and bounded performance cost. Concurrency level limited by number of kernel threads
    - uses **lightweight processes** (LWP) as user-level representation of kernel thread. Kernel schedules LWPs
- Two level model - same as many-to-many, but allows a user thread to be bound to a kernel thread.

Threads are executed in an interleaved way. For switching between threads we have two possibilities:

- Switching as kernel activity - *kernel* maintains execution state similar to process switching
- Handled by thread library - *library* maintains execution state, responsibility of programmer to call library to yield execution

Since creating threads dynamically is inefficient, we create a **thread pool** for an application with threads waiting for work. Allows faster request service, and puts a bound on the number of user threads per application.

2.3: Scheduling (slide 69+ for more detailed calculations)

Since only one task (process/thread/…) can access a HW resource at a time. **Processor scheduling** decides which **task** executed on each core at a time. There's quite a few processor scheduling policies, but we first need some terminology:

- **Decision mode**: when to make a decision
    - **Preemptive** (pause task when higher prio task shows up) or **non-preemptive**
    - **Time-based** (maintain a timer) or **event-based** (decision when some event occurs)
- **Priority function** defined what ready task to choose for execution, where ties are solved using **arbitration rules**
- **Waiting time** - time spent in the ready queue
- **Response time** - time until first response
- **Turnaround time** - time to completion
- **Throughput** - number of jobs completed per time unit.

First-Come, First-Served (**FCFS**) - (non-preemptively) schedule tasks in order of arrival
Shortest-Job-First (SJF) - (non-preemtpviely) schedule the task with the shortest next CPU burst. Optimal, but predict exec times.
Shortest-Remaining-Time-First - preemptive SJF, if a tasks comes in that has less remaining time, switch to this task
Round-Robin (**RR**) - each task is assigned a small unit of CPU time (quantum of length *q*), at the end of that quantum preempt it and add it to the end of the ready queue. Higher average turnaround time than SJF, but better *response time*. A task waits at most (n-1)q time units until it is given CPU time again. For large q, this is the same as FCFS, for small q we must ensure q is large w.r.t. context switch time as otherwise the overhead will be too high.

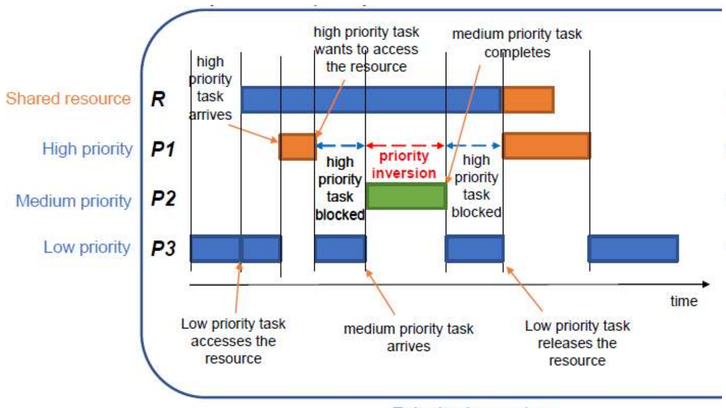
In **priority scheduling**, each task is assigned a priority number (either by the programmer or computed based on task properties). CPU is allocated to the process with highest priority, either preemptive or non-preemptive. A problem that can occur is **starvation**, where low priority processes may never get executed. This can be solved through **ageing**: increase the priority of the process over time. We discuss some examples:

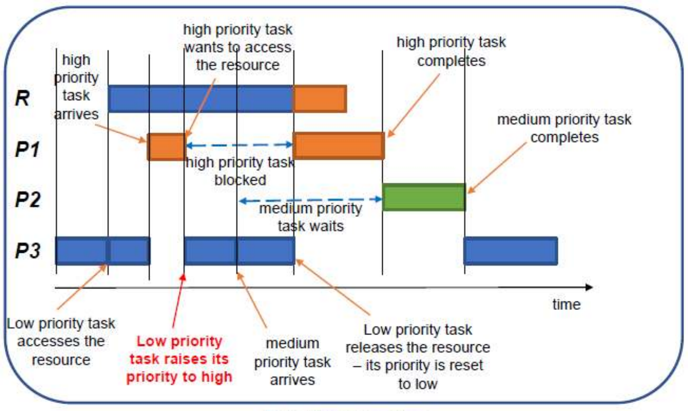
Rate Monotonic (RM): processes are often periodic in nature. A process with period T is activated every T time units, and its computation <u>must</u> complete within a relative deadline of D time units. Assign a higher priority to processes with a shorter period T, and execute in a preemptive fashion.
Deadline Monotonic (DM): Assign a higher priority to processes with a shorter <u>relative</u> deadline D, and execute in a preemptive fashion.
Earliest Deadline First (EDF): Highest priority is assigned to the process with the smallest remaining time until its deadline. So a process with shorter time remaining until its absolute deadline is assigned a higher priority. Execute in preemptive fashion.

We can combine different scheduling policies into a **multilevel queue scheduling** system, where we partition the ready queue in sub-queues of varying priority processes. Each queue has its own scheduling policy. For shceduling between queues, we have:
1. Fixed priority scheduling - serve high priority queue first → possible starvation
2. Time slice scheduling - each queue gets a certain amount of CPU time

We can allow tasks to move across sub-queues to implement ageing. The scheduler needs to know when to upgrade/demote a task, and a method to determine which queue a task initially enters.

Priority scheduling risks **blocking:** a low priority task obtains a resource, which causes a high priority tasks to have to wait on that task to finish executing its critical section.
Another problem is **priority inversion:** A middle priority task pre-empts the low priority task that holds the resource (so the low priority tasks still has the resource, but only continues after the middle priority task finished), so the high priority task must wait for the middle priority task and the low priority task's critical section.
⇒ **priority inheritance protocol**: priority of task P using the resource is <u>dynamically adjusted</u> to be the maximum of the priority of any other task that is blocked on the allocated resources of P *and* its own priority. Now, the middle priority task is the one that waits.


Shared resource
High priority
Medium priority
Low priority
R
P1
P2
P3
high priority task arrives
high priority task wants to access the resource
medium priority task completes
high priority task blocked
priority inversion
high priority task blocked
time
Low priority task accesses the resource
medium priority task arrives
Low priority task releases the resource
Priority inversion


R
P1
P2
P3
high priority task arrives
high priority task wants to access the resource
high priority task completes
medium priority task completes
high priority task blocked
medium priority task waits
time
Low priority task accesses the resource
Low priority task raises its priority to high
medium priority task arrives
Low priority task releases the resource – its priority is reset to low
Priority inversion

2.4: Preparational material
Interprocess communication (**IPC**) can be implemented either via **message passing** (small amounts of data) or **shared memory** (faster but difficult to implement). Message passing is especially useful in **distributed systems**. The **producer** sends messages through a message queue (either blocking, wait for receipt, or non-blocking). The consumer in turn reads messages from the queue either blocking (wait for a message to come in before cont.) or non-blocking. If both sender and receiver are blocking, we call it a **rendez-vous**, since the communicating processes synchronize.

For shared memory IPC, a process must request the kernel for some shared memory space in the *kernel memory*, making it addressable in user mode. Different processes can then link themselves to this shared space, directly reading/writing without further kernel involvement.

Lecture 4: Atomicity & Interference

4.1: Atomicity, traces & concurrency

An **atomic action** in an *indivisible* step in a program. A statement is atomic if it adheres to the **single reference rule**, making at most one reference to a shared variable. If we write <statement>, then the statement is always considered atomic.
A **trace** is a *sequence of atomic actions* in the execution of a program. Typically a sequence of *assignments* and *tests*.
A **concurrent trace** is a trace made of an *interleaving* of atomic actions of two or more tasks.

See the lecture slides for a complicated example of reasoning with traces

4.2: Synchronization

We want to coordinate the execution of programs in such a way that no two tasks ever modify a shared variable at the same time. Only one thread should be in the critical section at a time. Synchronization refers to *ordering the execution*.

There are several requirements for synchronization solutions (the last 4 only when concurrency is involved):

1. Functional correctness - satisfy specification (e.g., mutual exclusion)
2. Minimal waiting - only wait when correctness is in danger
3. Absence of deadlocks - no possible trace that ends up in a deadlocked state
4. Absence of livelock - ensure convergence towards a decision (do not just pass turns around doing no actual work)
5. Fairness in competition - prevent starvation of processes
   a. Weak - eventually, each contended is admitted to proceed
   b. Strong - we can put a bound on the waiting time of a contended

A famous example of a synchronization solution is **Peterson's algorithm** which combines different ideas (two processes only!):

- Take turns in crowded circumstances using a variable *t*
- Don't wait if there is no need (booleans bY and bX). We wait at most one execution time of the other process

For proving correctness, we can either use traces, or a **detailed annotation** (add assertions to the program) of the program with a proof by contradiction. A proof of the absence of deadline for Peterson's can be found on slide 35, using the failing assumption that both tasks are blocked on the while loop, and showing that this would result in contradicting states of the tasks.

4.3: Mutexes

The problem with Peterson's is that it only works for two tasks, and involves **busy-waiting** which wastes CPU cycles, and does not work for a single core with non-preemptive scheduling (as the waiting task will never release the processor).
The OS provide **semaphores** and **mutexes** to achieve mutual exclusion, where a mutex is a specialized variant of a semaphore. A mutex is a binary semaphore, initialized to the value 1. A mutex supports two atomic operations:

- lock(m): <await(m>0); m→
- unlock(m): <m++>

4.4: Preparational material

The problem with non-atomic statements is that old copies of shared variables can be stored in internal registers or in memory locations. A possible trace is a trace in which all tests yield true, denoted (boolean expression).

Lecture 5: Action synchronization

We prove program properties using **invariants**: assertions that hold at *all* control point of a program. A **topology invariant** is an invariant derived directly from the code, comparing the number of times certain operations are executed, or the value of a variable based on the number of times a set of actions gets executed. To do so, we give names to operations and introduce counting variables.

See example and exercise on slides

Action synchronization can be achieved using semaphores. We can protect shared data with a mutex, and use a semaphore to also prevent reading from an empty buffer and writing to a full buffer. The semaphore can indicate how much of a resource is available. using semaphores is an alternative to **busy-waiting**: repeated testing without going to sleep. Busy-waiting can be used only when the waiting time is guaranteed short, and there's nothing else to do. It only works if the test using *await* and locking the mutex are executed atomically: otherwise the condition can become false *after* the check, but *before* execution.

A semaphore is a <u>non-negative</u> integer with some initial value, and two atomic operations:

- P(s): <await(s>0); s := s-1>
- V(S): <s := s+1>

The basic action-synchronization problem is as follows:

Given:

- Collection of tasks executing a collection of actions A_i and B_j for *i,j* = 0,1,...
- Synchronization condition (invariant) of the shape $\sum_i a_i \times cA_i \leq \sum_j b_j \times cB_j + e$ for non-negative constant a_i, b_j and e

Solution:

- Introduce semaphore *s*, with initial value s_0 = e
- Replace action A_i with $P(s)^{a\_i} A_i$
- Replace action B_j with $B_j V(s)^{b\_j}$

Fairness of the solution is the ame as the fairness of the semaphores and mutex, and is therefore dependent on their implementations. Functional correctness and minimal waiting are ensured by construction.

A **deadlock state** is a system state in which a <u>set of tasks</u> is blocked indefinitely: each task in the set is blocked on another task in the same set. Absence of deadlocks is typically proven by contradiction: assume deadlock, investigate all task sets that can be blocked at the same time, and show a contradiction for all possible combinations of blocking actions of those tasks. When a deadlock is identified, a full proof should be given by providing a valid trace!

Some ways of preventing deadlocks are:

- Always calling unlock(m) after lock(m) - let critical sections terminate
- Avoid **cyclic waiting** - avoid P or lock operations that may block indefinitely between lock(m) and unlock(m)
    - Solution: use a fixed order when calling P-operations on semaphores or mutexes
- Avoid greedy consumers - $P(a)^k$ should be an indivisible atomic operation when tasks compete for limited resources
    - Otherwise one task may do k/2 P(a) operations, then another also does k/2, now both are stuck

5.2: Preparational material

If A is an action in the program, cA denoted the number of completed executions of A. Value of variables can be expressed using these via invariants, and to prove an invariant over a variable you can replace the variable by such 'action-counting invariants'. A Semaphore has two inherent invariants (non-negative and value determined by initial + cV - cP), which can be combined into one cP ≤ s_0 + cV.
The **progress property** states that blocking is only allowed if the safety properties would be violated.

Lecture 6: Condition synchronization

6.1: Condition variables

We require **condition synchronization** to enforce properties we cannot check just by counting actions, implemented using **condition variables** or **monitors**. We require explicit communication (signalling) between tasks.

- When a condition may be violated: check and block
- When a condition may have become true: signal waiters

To do so, we have the operations *Wait(..., cv)*, *Signal(cv)*, *Sigall(cv)*, *Empty(cv)*

- *Wait(m,cv)* is short for *<unlock(m); Wait(cv)>; lock(m)* - atomic, if preempted after unlocking, may miss the signal and wait
- Wait() can be extended with a **timeout**, which may increase robustness agains programming errors.

We generally follow the same steps:

1. Identify critical sections and protect them (accessing the same data in multiple tasks)
2. Identify where a condition may become false, and add a guard that prevents breaking the invariant when cont.
    a. If we need x ≥ 0, and we have <x := x-10>, add the guard *while(x<10) Wait(m, cv)*
3. Identify where a condition may become true, and signal one or more waiters
    a. Whether to use *signal* or *sigall* depends on the operation executed, and the program context

To find the guard condition:

1. Take the synchronization invariant
2. Compute preconditions for the statements that might disturb the invariant
    a. Substritute disturbing statement in the synchronization invariant
3. Negate the resulting condition

Do the practice question on the slides

6.2: Monitors

A **monitor** is an object containing data and operations on those data. Procedures of a monitor are executed under mutual exclusion, as at most one task can be ‘inside the monitor’. Synchronization is implemented using condition variables.

We can use the monitor to synchronze accesses to read and write actions. The monitor then keeps track of how many tasks are reading/writing at this point in time (keeps track of system state). We implement entry and exit protocols in the monitor, and place those around the read/write actions in the tasks themselves.

The **signaling discipline** used determines what task is inside the monitor right after the signal. Correctness depends on the choice:

- Signal & exit
    - The task executing a signal exits the monitor immediately (does not support *sigall*)
    - B(cv) certainly holds after *wait()*
- Signal & continue
    - The task executing a signal continues to use the monitor until a *wait* or reaching the end
    - Condition may or may not hold after *wait()*
- Signal & wait
    - Task executing a signal is stopped in favour of the signalled process, and competes with all other processes again
    - B(cv) certainly holds after *wait()*
- Signal & urgent wait
    - Similar to wait, but signaller has priority over other tasks to access the monitor again after the *signal*.
    - B(cv) certainly holds after *wait()*

6.3: Preparational material

Counting is not sufficient when the value of a variable is updated using another variable, or by means of a multiplication. Additionally, a disjunction in a cv cannot be maintained using action synchronization. Each condition variable cv is associated with a condition B(cv).

Lecture 7: Deadlocks

A task is **blocked** if it is *waiting* on a blocking synchronization action. A **deadlock** occurs when a set of tasks is blocked indefinitely and cannot make progress anymore. A set of tasks D is **deadlocked** if

- All tasks in D are blocked or terminated
- There is at least one non-terminated task in D, and
- for each non-terminated task T in D, any task that might unblock T is also in D

To analyse deadlocks, we need to distinguish two types of resources:

- **Consumable resources**: taken away upon use, so the number of resources varies (producer/consumer problems)
- **Reusable resources**: given back after use, so the number of resources is fixed (reader/writer problems)

7.1: Analysis of deadlocks

To analyze consumable resources (and condition sync) we use **wait-for graphs**. An edge p1 → p3 means that task p1 may unblock p3 by falsifying the blocking condition (edge label). The wait-for-graph represents a deadlock state if and only if (1) there is a cycle in the graph, and (2) no task outside the cycle can unblock a task in the cycle.
To prove absence of deadlocks, repeat the following for *every action that may block a task*:

1. Assume one task T is blocked on a certain action, and add it to the graph
2. Check which other task(s) can unblock T and under which condition, add these to the graph and connect them to T
3. Check whether each task T' can possibly be blocked, repeat from (1)
4. Once the graph is built, check for deadlock state
    a. No → contradiction
    b. Yes → build a trace showing how we reach said sate.

To analyze reusable resources (and action sync) we use **dependency graphs**, showing which resources are requested/held by which task. A task is blocked if it has an outgoing edge that is not directly removable, so for which the requested resources are not all free. **Reduction** by repeatedly removing all non-blocking tasks. Deadlock state if the reduced graph is not empty (**knot**). This issues that all tasks always eventually release all the resources they've acquired.

To show absence of deadlocks, we must show the graph are reducible in all dependency graphs we can generate by allocating the resources according to program code: examine all reachable states of the FSM corresponding to the program.

7.2: Dealing with deadlocks

Three options:

- Prevention - done at design time
- Avoidance - dynamically check to avoid entering risky states (expensive)
- Detection & recovery - check only whether we are in a deadlock state or not, and try to recover

7.2.1: Prevention:

- Make the reduces dependency graphs empty by construction, prevent cycles in wait-for graphs
- Prevent circular wait, ensure terminating critical sections, acquire all resources at once
- Allow preemption of resources when needed

7.2.2: Avoidance:

- Upon execution of a potentially blocking action, check if an open execution path exists (otherwise deny or postpone)
    - Means the **reduces max claim graph is empty**
    - Postpoining works if blocking action refers to reusable resources and we can compute possible future states
- e.g., **Banker's algorihtm** ⇒ **See lecture slides 45+**

The **max claim graph** includes the (maximum) *future resource claims* using dashed arrows.
A state is called **open** for task T, if all resources can be given directly to that task.
A state is called **safe for a task**, if this task can be given its maximum number of resources eventually. (dep-graph reduction)
A state is called **safe** if it is safe for all tasks.

If a state is safe for some task T, granting the resources to T will lead to a state at least as safe as the current state.

7.2.3: Detection & Recovery:

Invoke a detection algorithm periodically to check if deadlock occurs: examine the state upon execution of a blocking action, and an algorithm to recover from a deadlock. Causes large overheads. Recovery can be done:

- Locally - inside the blocked task (e.g., release all resources)
- Globally - recovery policy
    - Kill tasks in the deadlock set (all or based on criteria)
    - Roll back to safe state, only if alternative execution path exists, requires recording of state at checkpoints
    - Preempt resources - dealocate resources from tasks in deadlock set (not always possible)

Lecture 8: Virtual memory (part 1)

Ideal memory is: simple, private, non-volatile, fast to access, unlimited in capacity and cheap. To approach this, we use a **memory hierarchy** and **virtualization**.

8.1: Memory management in early systems (more details in preparational material)

Every active process resides in its entirety in MM, allocated a contiguous part. Lead to three partitioning schemes:

- **Fixed** partition sizes - processes assigned to a free partition of sufficient size
  - # of active processes limited by # of partitions, and size of processes limited by size of largest partition
  - Memory waste due to **internal fragmentation** (unused space in partition)
- **Dynamically** allocate space in MM
  - # of processes only limited by memory size, same goes for process size
  - Memory waste due to **external fragmentation** (unused space MM)
- **Relocatable** - same as dynamic, but can compact memory
  - compaction is costly and requires dynamic address binding

8.2: Memory Paging

With memory paging, programs do not have to be stored contiguously in MM. We divide the process address space in segments of identical sizes called **pages**, and divide MM in segments of identical sizes called **page frames**. Now, every free *frame* is always usable, so we no longer have external fragmentation. Internal fragmentation is limited to the size of a single page (program size varies).

8.2.1: Keep track of page locations

Solution 1: The OS maintains a **page table** for each process, the address at which the table is stored is saved in the process' PCBs. The page table records the frame used by each page (page-frame mapping). While this makes it easy to find a page in MM, it consumer a lot of memory. Note that we do not need to store the pageID, as this corresponds to the index of the table.

Solution 2: The OS maintains single **inverted page table** (**frame table**), recording which page of which process is loaded in which frame. Memory consumption is now constant, but finding the location of a page is more time consuming (iterate over all entries).

8.2.2: Address binding (translate logical addresses into physical addresses)

Use **virtual addresses** in the program code, and translate these to physical addresses during execution. A virtual address consists of $|p|$ bits encoding the page number, and $|w|$ bits encoding the offset within the page in number of words. This can refer to $2^{|p|}$ pages, and the page size is $2^{|w|}$ words/page. Virtual memory size is then $2^{|p|+|w|}$ words.
Physical addresses are structured the same, but instead using $|f|$ bits to encode the frame number. Virtual memory size seen by each process can be larger or smaller than the physical memory size.

If we are using page tables, reading from/writing to memory requires two memory accesses. One for accessing the page table and one for accessing actual data. A **Page Table Base Register** holds the value of PT for fast access to the page table.
If we are using frame tables, reading from/writing to memory requires up to as many memory accesses as there are entries in the frame table (check pid and page number for each entry).

A **Translation Look-Aside Buffer** (**TLB**) is a small cache memory that keeps track of the locations of the most recently used pages in MM. It accelerates address translation and thus memory accesses. Only contains the frame id in which the most recently accessed pages are loaded. Whenever a page is accessed, first check if it is in the TLB, if not, use the page table. If it still can't be found, generate a **page fault** as the page is not yet in PM.

8.3.3: Running partially loaded processes

Use **demand paging** to only bring a page into MM when it is actually needed, so we no longer need the entire process in MM. Some parts of code are only required in very specific cases (error handling, user choice). This gives the appearance of an infinite PM.

8.4: Preparational material

The OS places frequently used data higher in the hierarchy. Updates are first applied to upper memory. Upward moves are copy operations, while downwards moves are destructive.

We keep track of holes in MM using a list, and allocate via different possible schemes:
First-Fit allocation: allocate the *first* partition that is big enough. This is fast but wastes more memory space
Best-fit allocation: allocate the smallest partition fitting the requirements. Leaves smallest leftover partition, but takes more time. Not the best for dynamic partitions, as it results in excessive memory fragmentation due to smaller free space.
Next-Fit (rotating first-fit): start search where previous search stopped
Worst-fit: always allocate largest available partition

When using dynamic partitioning, after ddealocation we try to combine free areas of memory. Ensure we only de-allocate processes that are really idle. For **compaction**, we relocate programs in memory so that they're contiguous, adjusting every address and reference to an address within each program (does not work for static address binding). Different schemes are:

(1) complete compaction, (2) partial compaction, (3) minimal data movement

To keep track of where each process is w.r.t. its original location, we use a **relocation register** (value to be added to each address referenced in the program) and a **limit register** (size of the memory space accessible by the program).

Swapping must be done when there is insufficient space in MM to load a program. We move something else to 2nd storage to make room. We swap out only those parts of process space that are not already on disk. Process is either swapped to an arbitrary file, or a special partition on disk (swap space).

Lecture 9: Virtual memory (part 2)

9.1: Demand paging

We first check whether the page is in MM using the TLB or page table. If both checks fail, a **page fault** occurs and we check whether the page is actually in the address space of the process (if not, terminate the process). We then look for an empty frame in MM, and either load the page into this free frame or swap out a different frame. Then we update the page tables and translate the VA.
Note that a single memory reference may create several page faults (i.e., when the page table is not in MM yet)

If *p* is the page fault rate, then the **Effective Access Time** =

(1-p) * memory-access-time + p * (page-fault-overhead + swap-page-out + swap-page-in + restart-overhead)

To reduce the page fault rate we can use **pre-paging**: preload those pages predicted to be accessed soon.

9.2: Page replacement and load strategies

We must choose what page to swap out upon a page fault, while preventing **trashing**: replacing in-use pages by other active pages. This way, a process would spend more time treating page faults than execution program code. Three approaches exist, of which we look at the details for two of them. The third approach assigns each process a fixed number of frames based on program properties, then we can e.g., swap out only pages from the page loading a new frame, or any with a lower priority.

We first look at **global** approaches: when we must swap out a page, select *any* page in MM according to:

- **MIN** replacement - select the page that will not be used for the longest time *in the future* (not possible in reality)
- **Random** replacement
- **FIFO** replacement - select the page that has been in MM for the longest time out of those loaded
- **LRU** replacement - select the page that is *least recently used*
- **Second chance** replacement (clock replacement) - circular list with use-bit u, set u to 1 upon reference, search clockwise for first page with u=0, while setting the use-bit to 0.

Comparisons are done through **reference strings** (execution trace with only memory references). We compare the # of page faults.

We also have the **working set** approach, using **time locality** of programs: the set of accessed pages remains rather constant for short periods of time. We keep only the pages of the past *tau* memory references by each process in MM. Working set at time *t* is given by:

$W(t, \tau) = \{r_j \,|\, t - \tau < j \le t\}$ for reference string $r_0 r_1 \ldots r_T$

Since this is a set, duplicates are removed (hence the size of the working set varies over time). If tau is too small, we have thrashing. If tau is too large, fewer processes can fit in MM. If the sum of the working set sizes exceeds MM size, we remove one process from main memory (many ways to choose such a process: priority, last activated, smallest, largest).

9.3: Segmentation

A **segment** is a logical block from a program. Whenever a program needs data or instruction from a segment, the whole segment is loaded into MM. Each segment is further divided into pages, so that VA = segID pageNR offset. We maintain a segment table for address translation. Each memory reference requires 3 accesses to MM, which can be reduced using the TLB. A balance is made between #segments and #pages.

Advantages of VM:

- Process size not restricted by MM size
- Memory used more efficiently, eliminating external fragmentation and reducing internal fragmentation
- Placement of a program in memory need not be known at design time
- Facilitates dynamic linking of program segments
- Allows sharing of code and data by sharing access to pages

Disadvantages of VM:

- Increased processor HW costs
- Increased overhead for handling page faults
- Increased software complexity to prevent thrashing

Lecture 10: File system

A **file** is a logical data storage unit, providing abstraction from physical properties of storage devices (via an API and system programs). The **file system** provides an abstract view of files & directories, and provides *access control* and other *security* measures.

10.1: User view of the file system

For users, a file is a chunk of related data. Logical file organization can either (1) view files as named byte sequences (type is only a naming convention), where the file system is unaware of content & structure, or (2) files structured based on type, where the file type determines the file's use and structure.
Files have **permanent** (type, owner, size, access rights, dates) and **temporary attributes** (read/write pointer, buffers).

Files can be accessed in different ways (also dependent on hardware).

1. Sequential access: read all bytes from the beginning - only allows rewinding
2. Direct access: read bytes in any order - read can involve moving (seeking)
3. Access through index files: built on top of direct access, construct an index of the file
    a. First search the index, then use the pointer to access the file directly.

The file system can be interpreted in two ways:

1. **Physical file system** - data structure on storage device representing collection of files.
2. **Virtual file system** - OS subsystem dealing with files, API to access and modify such files.

Any resource (and devices) is seen and manipulated as a file, while these special 'device files' do not exist in the PFS.

A **directory** is a file containing information about other files (and directories). A **disk** can be subdivided into **partitions** (hardware portion), which can involve multiple disks. A **volume** is a logical drive mapped to one or more partitions. Every directory <u>must</u> have a parent directory, so deleting a directory also requires deleting all contained files first. File deletion is a *directory operation* (we remove the entry reference) so recovery of files is possible as long as memory is not overwritten.

Single-level Directory: all files in one directory, all file names must be unique, no possibility of grouping data.
Two-level Directory: separate directory for each user, allows for more efficient searching (first user dir, then system dir)
Tree-Structured Directories: allows for grouping of files per user, efficient searching
Graph directories: either acyclic or general. Shared subdirectories and diles have links as directory entries. When allowing cycles, by-pass links during search.

When calling *open(path)*, the kernel recursively opens directories, creates new **descriptors** in the **Open File Table** storing dynamic attributes such as read/write pointers, and caches the reference obtained. The pointer to the descriptor is returned to user process for later reference (so subdirectories are also added to the OFT). So the *open* and *close* call create and initialize/destroy *file descriptors*.

File owners can control who can do what for their file, by defining access modes for the three classes of users (owner, group, public). This is done by setting the read (R), write (W), execute (X) bits for each class (bit order RWX translated to decimal between 0 and 7).

10.2: OS view & implementation

The File system has a layered structure: (1) Directory Management, (2) Basic File System, (3) Device Organization methods
Directory Management tasks are mainly **name resolution**: file descriptor corresponding to symbolic file name. **Mounting** makes files accessible to the user via the file system. The eventual file description is passed to the basic FS for opening.

Basic File system tasks are opening and closing files, and verifying access rights upon requests. It maintains an OFT, and upon opening a file, its descriptor is copied to the OFT (i-node table per file). The **Open File ID** is a pointer ot an OFT entry, and is returned to directory management, which returns it to the application.

Device organization level tasks are management of physical blocks (mapping logical files to physical blocks). Regards the device *logically* as sequence of blocks. The **blocking factor** is the ratio between logical records and physical blocks. File organization on disk:

- Contiguous - not flexible due to external fragmentation
- Linked:
    - Simple linked blocks - bad performance as random access is difficult, must traverse the blocks in sequence
    - Using dedicated table of pointers (1 per block) - slight improvement
    - Linking groups of adjacent blocks instead of individual blocksIndex:
    - Start with block of block indices (fixed size index table)
    - Linked list of index blocks (multilevel hierarchy)
    - Block with (start, #blocks) entries

UNIX V7 has for each file descriptor 13 entries mapping to physical blocks, where the last 3 are indirection blocks (single/double/triple) of 128 entries. For a total of ($10+128+128^2+128^3$) blocks.

10.3: Distributed File Systems
We want a unified view of all files to users, hiding the distributed structure and supporting sharing. An important issue here is server performance:

- **availability**: probability server is accessible
- **reliability**: expected time til server failure
- **scalability**: ability to grow with number of clients and requests
- **access time**

Decisions to make are **global**/**local** (is there a single directory structure seen by all, or each user a local view), is the location **transparent** (does the path name reveal hosting machine), and do we have **location independence** (must the pathname change when a file is moved between machines).

Different sharing policies are:

1. Same as single machine: effectuate writes as soon as possible - unpredictable delays
2. Value/result or session semantics: copy upon open, write upon close
3. Transaction semantics: reduce time until changes are visible, atomic modification
4. Immutable-files: files are read-only, writing a file creates a new one

Both the client and the server have a file system. The server FS serves requests from the network, and the client FS servers requests from the application and transparently includes server FS. A choice has to be made on what level the connection client-server exists:

1. Application level: client accesses the remote FS as just another application
2. Basic FS level: **stateful** server. Client uses the remote FS as a process, server maintains OFT.
3. Device organization level: **stateless** server. Client uses the remove FS as a block storage, and maintains its own OFT.

Lecture 11: I/O management

11.1: I/O Device Controllers

An I/O device communicates with the CPU through hardware registers available in a **device controller**. We call this memory-mapped I/O. Provided information on device status, and requests made by the CPU. Bytes transferred to and from the I/O device are buffered. The device controller can be accessed in two ways:

1. CPU instruction set extended with special instructions. No physical address associated to reach register, so user processes cannot directly access the device.
2. Physical address space extended with device registers. The device can then be mapped to user space via virtual addresses, but virtual memory management is a bit more complex.

11.2: I/O subsystem

This is the part of the OS that manages I/O devices. It presents an abstract view, facilitates sharing and optimizes performance. It handles the large variety in I/O devices, with different (brand-)specific approaches. Additionally, we want to be able to add new devices even after OS development and installation.

Abstraction happens at two levels, with an additional level inbetween (for generic code handling classes of I/O devices - sequences of high-level operations and implementation of protocols).

High-level abstraction: divide I/O devices in different (3) classes

- Block-oriented or storage devices: data in arbitrary order, action sequence depends on operation and data location
- Stream-oriented devices - optimized for single byte access
- Network communication devices: provide connection and communication protocols

Obviously this does not cover all, which can be solved by:

1. Let user applications directly access drivers of other device groups, which makes code device-specific and low level.
2. Extend OS capabilities with new APIs via external libraries to provide vendor-speicifc abstraction.

Low-level abstraction: **device drivers** implement collections of standard operations. The communication between driver and controller can happen as follows:

1. **Polling**: driver busy-waits for device controller - only acceptable when I/O is known to be fast vs context switching
2. **Interrupts**: driver initiates and then yields the CPU. Controller informs the CPU by means of an interrupt
3. **DMA**: after initialization, DMA independently moves *groups of data* between device controller and MM. Less overhead for the CPU as only one interrupt need be handled when the whole transfer is completed.

11.3: I/O buffering

We require buffering to solve several potential issues for processes waiting/suspending until an interrupt from I/O device:

1. Speed/latency mismatch: process must wait for relatively slow I/O
2. Data granularity mismatch: application may expect to receive data in smaller pieces than a block
3. Conflict with OS swapping decisions: pages containing the VA range must remain in physical memory until I/O completes, to prevent writing data at the wrong *pa*.
4. Risk of single process deadlock: process issues I/O command, then is swapped out before I/O operation starts

The **buffer** is dedicated kernel memory space or registers that hold data of a producer until its consumer is ready. It *decouples produces and consumer*.

- Resolves speed & granularity difference
- Resolves swapping issues: buffer remind in PM permanently
- Improves efficiency: predict future needs, delay outputs until the right time (more at once), cache data

Performance is evaluated by **throughput** - how many data transferred per second - and **latency** - delay from data generation until data processing. T: time to transfer one data in MM. M: time for moving from kernel to user. C: process time per data. D: total time until data available to user. Throughput is then equal to 1/D. Buffering can be done in alternative ways:

- **No buffer**
  - D = T + C
- **Single buffer**: either of input or output. Enables **asynchronous transfer** and allows process swapping.
  - D = max(C,T) + M
- **Double buffering** (buffer swapping) reduces idle time: OS can move content of one buffer to use space while the other buffer gets filled.
  - D = max(M+C, T)
- **Circular buffering** can better handle burts of data
  - Same throughput

Performance is evaluated by **throughput** - how many data transferred per second - and **latency** - delay from data generation until data processing.

11.4: Disk Scheduling

Hard-drive is divided in several **platters**, which are divided in **tracks**, which are divided in **sectors**. A **cylinder** is a set of tracks at the same arm position (same radius). We can move the read-write head forward and back, and rotate the spindle to rotate all platters.

Average time to read/write one block on the disk is

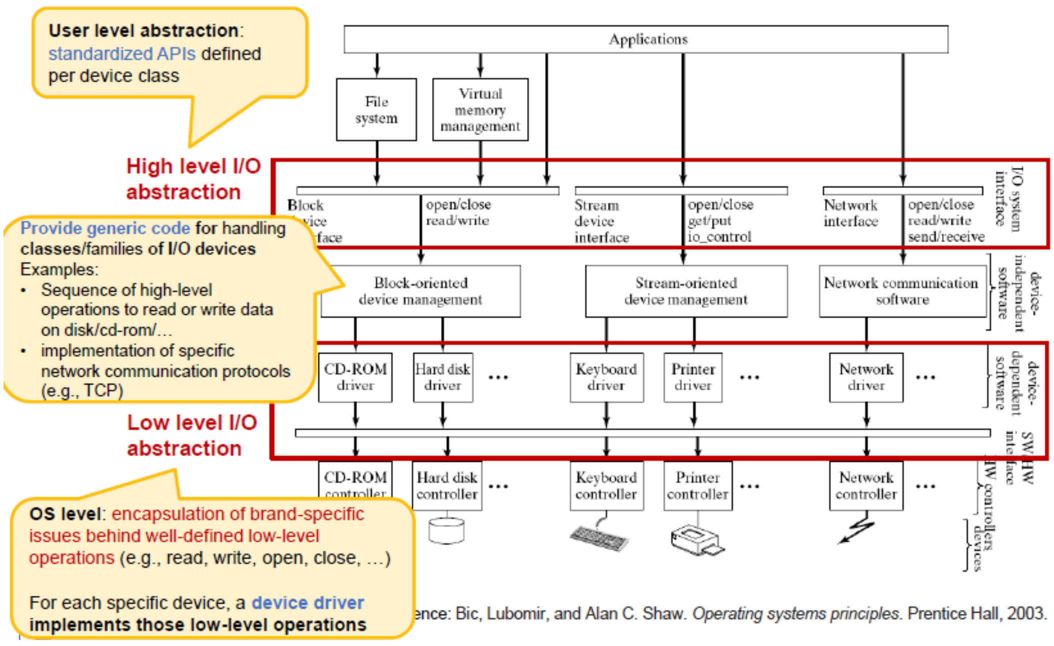
$$T_{read} = T_{Seek\ cylinder} + T_{move\ to\ sector} + T_{read\ block} = T_{seek\ cylinder} + 1/2r + b/rN$$

where r = rotations per second, N = #bytes per track, B = #bytes per sector.

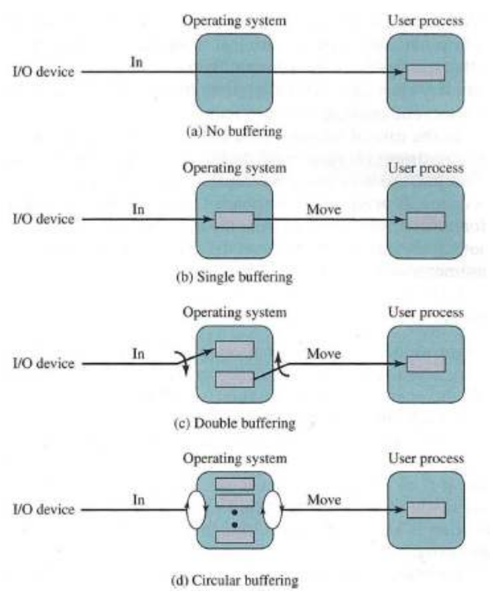
Moving the read-write head is the slowest operation, so the OS must optmize disk head movement. Therefore, we first *buffer* requests for blocks, and serve these request according to a scheduling policy:

- **FIFO**: treat requests in the order they are emitted
- **Shortest Seek Time First**: treat the buffered request requiring the smallest head movement ⇒ risks starvation
- **Elevator scan**: complete full swing of the head in either direction (NOTE: swing here is forward/backward, so limited)
    - Changing the direction of the head swing may be costly

We can compare the total distance traveled in #cylinders, via a reference string containing only cylinder numbers (since finding tracks and sectors is way faster compared to finding the cylinder).


User level abstraction: standardized APIs defined per device class
High level I/O abstraction
Provide generic code for handling classes/families of I/O devices
Examples:
• Sequence of high-level operations to read or write data on disk/cd-rom/…
• implementation of specific network communication protocols (e.g., TCP)
Low level I/O abstraction
OS level: encapsulation of brand-specific issues behind well-defined low-level operations (e.g., read, write, open, close, …)
For each specific device, a device driver implements those low-level operations
Applications
File system
Virtual memory management
Block device interface
open/close read/write
Stream device interface
open/close get/put io_control
Network interface
open/close read/write send/receive
I/O system interface
Block-oriented device management
Stream-oriented device management
Network communication software
device-independent software
CD-ROM driver
Hard disk driver
Keyboard driver
Printer driver
Network driver
device-dependent software
SW/HW interface
CD-ROM controller
Hard disk controller
Keyboard controller
Printer controller
Network controller
HW controllers
devices

ence: Bic, Lubomir, and Alan C. Shaw. *Operating systems principles*. Prentice Hall, 2003.



(a) No buffering

(b) Single buffering

(c) Double buffering

(d) Circular buffering